

Semantic-based Code Obfuscation by Abstract Interpretation

Mila Dalla Preda and Roberto Giacobazzi

Dipartimento di Informatica, Università di Verona
Strada Le Grazie 15, 37134 Verona (Italy)
dallapre@sci.univr.it | roberto.giacobazzi@univr.it

Abstract. In this paper we introduce a semantic-based approach for code obfuscation. The aim of code obfuscation is to prevent malicious users to disclose properties of the original source program. This goal can be precisely modeled by abstract interpretation, where the hiding of properties corresponds to abstract the semantics. We derive a general theory based on abstract interpretation, where the potency of code obfuscation can be measured by comparing hidden properties in the lattice of abstract interpretations. Semantic-based code obfuscation is applied to show that well known program transformation methods, such as constant propagation, can be seen as code obfuscation.

Keywords: Code Obfuscation, Abstract Interpretation, Program Transformation, Semantics.

1 Introduction

Code obfuscation is a program transformation typically intended to prevent reverse engineering [2, 4–6]. A number of results are known in the literature providing obfuscation algorithms such as: *Layout* transformations, which remove source code formatting and scramble identifiers; *control* transformations, which affect the control flow of the program; and *data* transformations, which operate obfuscating the data structures used in the program [5]. The major negative result on code obfuscation is given in Barak *et al.* [1]. They prove that there is no obfuscation method that works for any program and it is able to transform them in such a way that the only properties which can be disclosed are those which can be derived from the input/output semantics. This result is not as bad as it might seem, in fact even though the “ideal” obfuscator of Barak *et al.* does not exist, software obfuscation would be still useful when employed for hiding specific code properties and working for specific classes of programs [14]. The classical notion of code obfuscation of Collberg *et al.* [4–6] defines an obfuscator as a potent transformation that preserves the *observable behavior* of programs. In this setting a transformation is *potent* when the obfuscated program is more complex than the original one. Clearly this definition of code obfuscator relies on a fixed metric for measuring program complexity, and finding such metrics is a major challenge in practical code obfuscation algorithms.

The problem

The major drawback of most software obfuscation techniques is that they do not have a well found theoretical base, and thus it is unclear how effective they are. Even if semantic preservation is implied in code obfuscation [18], the lack of a complete formal setting where these program transformations can be studied defects any possibility of comparing them with respect to their ability to obfuscate program properties. The main problem here is to fix a measure for potency. Usually syntactic (textual) measures are considered, such as code length, nesting-levels, fan-in-out complexity, branching, *etc* [5]. Semantic-based measures are instead less common, even though they may provide a deeper insight in the true potency of code obfuscation. In order to understand this point we need to model attackers, *i.e.*, code de-obfuscation techniques. Static program analysis is the standard method for making reverse-engineering. Recently dynamic attacks have also been considered in [3] for strengthening static ones for de-obfuscation. Both static and dynamic attacks strongly relies upon program semantics: the first corresponds precisely to a decidable semantic abstraction, while the last are based on the concrete semantics, *e.g.*, interpreters. In both cases syntactic measures can be misleading. More significant measures have to be derived from semantics and this, as far as we know, is an open problem.

Main results

In this paper we consider the lattice of abstract interpretations as the domain for measuring potency. The goal of code obfuscation is to prevent reverse engineering (*e.g.*, by static or dynamic analysis) to grasp sensible properties on program's structure and semantics. It is well known that static analysis can be completely and fully specified as an abstract interpretation [10], *i.e.*, as an approximation of the concrete semantics of programs. Similarly, dynamic analysis can be seen as a possibly non decidable approximation of the concrete semantics. In this sense, code obfuscation can be seen as a way to prevent that some information about program behaviour is disclosed by an abstract interpretation of its semantics. In order to apply abstract interpretation as a model for attackers, we need to replace syntactic code obfuscators with corresponding semantic transformations. Recently, Cousot & Cousot in [12] have introduced a semantic-based formalization of program transformation based on abstract interpretation. In this construction the relation between syntactic and semantic transformations is specified by an abstract interpretation, where syntax is an abstraction of semantics. This allows us to mirror code obfuscators, viewed as syntactic transformations, in the semantics, by considering corresponding semantic obfuscators instead. Therefore the role of abstract interpretation in a semantic-based approach to code obfuscation is crucial: By fixing a formal relation between syntactic and semantic transformations, abstract interpretation provides the most general setting where attackers can be compared by comparing abstractions, leading us to derive a semantic-based metric for the potency

of code obfuscation. Traditionally code obfuscation is intended to preserve input/output (denotational) semantics of programs. This is again an unreasonable restriction: Semantics at different levels of abstraction can be related by abstract interpretation in a hierarchy of semantics [9]. Therefore, any program transformation τ which preserves a given semantics in the hierarchy may act as a code obfuscation for those properties that are not preserved by the transformation. The idea is that the transformed program $\tau[P]$ is more complex (obscure) than the original program P , *i.e.*, τ is potent, when there exists a semantic property, *i.e.*, an abstract semantics, which is not preserved by the transformed program $\tau[P]$. Potency is therefore strictly related with the rate of abstraction of the most concrete preserved semantics of a code transformation. This corresponds to map code transformations to the lattice of abstract interpretations to measure their obfuscating potency. We introduce a constructive systematic method for deriving the most concrete preserved abstraction of a generic code transformation. Then we generalize Collberg *et al.* definition of code obfuscation [4, 5] by considering as obfuscators those transformations that mask some abstractions in the lattice of abstract interpretations. This means that, in principle, *any program transformation may potentially act as a code obfuscator*. We show how a well known program transformation for solving the constant propagation problem, can be seen as a code obfuscator. In this case the transformation acts as a code obfuscator relatively to the command-line structure.

2 Preliminaries

Abstract Interpretation. The notation $\langle C, \leq, \vee, \wedge, \top, \perp \rangle$ denotes a complete lattice C , with ordering \leq , *lub* \vee , *glb* \wedge , greatest element \top , and least element \perp . $x \in C$ is *meet-irreducible* if $x = a \wedge b$ then $x \in \{a, b\}$. The set of meet-irreducible elements in C is denoted $Mirr(C)$. The downward closure of $S \subseteq C$ is $\downarrow S \stackrel{\text{def}}{=} \{x \in C \mid \exists y \in S. x \leq y\}$, and for $x \in C$, $\downarrow x$ is a shorthand for $\downarrow \{x\}$, while the upward closure \uparrow is dually defined. Abstract domains can be formulated either in terms of Galois connections or upper closures operators [11]. An *upper closure operator* on a ordered set C , $\mu : C \rightarrow C$, is a monotone, idempotent, and extensive ($\forall x \in C. x \leq \mu(x)$) function. With $uco(C)$ we denote the set of all upper closure operators of C . Each closure $\mu \in uco(C)$ is uniquely determined by the set of its fix-points $\mu(C)$ [17]. $X \subseteq C$ is a set of fix-points of an upper closure operator on C iff X is a *Moore family* of C , *i.e.*, $X = \mathcal{M}(X) \stackrel{\text{def}}{=} \{\wedge S \mid S \subseteq X\}$ - where $\wedge \emptyset = \top \in \mathcal{M}(X)$. If C is a complete lattice also $uco(C)$ is a complete lattice denoted by $\langle uco(C), \sqsubseteq, \sqcup, \sqcap, \lambda x. \top, \lambda x. x \rangle$ where, given two closures η and μ , $\eta \sqsubseteq \mu$ iff $\mu(C) \subseteq \eta(C)$. On the other side a concrete domain C and an abstract domain A form a *Galois Connection* (GC), denoted by (C, α, A, γ) , when $\alpha : C \rightarrow A$ and $\gamma : A \rightarrow C$ define an adjunction, namely $\forall x \in C, y \in A : \alpha(x) \leq y \Leftrightarrow x \leq \gamma(y)$. Given a GC (C, α, A, γ) the closure corresponding to the abstract domain A is $\rho = \gamma \circ \alpha$ on C . If ρ is a closure in C and $\iota : \rho(C) \rightarrow A$ is an isomorphism of complete lattices (with inverse ι^{-1}), then $(C, \iota \circ \rho, A, \iota^{-1})$ is a GC. Given a function $f : C \rightarrow C$ and a closure $\alpha \in uco(C)$, then $f^\#$ is a *correct (sound)*

Arithmetic expressions	$E \in \mathbb{E} \quad E ::= n \mid X \mid E_1 - E_2$
Boolean expressions	$B \in \mathbb{B} \quad B ::= E_1 < E_2 \mid B_1 \vee B_2 \mid \neg B_1 \mid true \mid false$
Program actions	$A \in \mathbb{A} \quad A ::= X := E \mid X :=? \mid B$
Commands	$C \in \mathbb{C} \quad C ::= L_1 : A \rightarrow L_2$ where $L_1, L_2 \in \mathbb{L} \cup \{ \perp \}$ and \perp is the undefined label

Table 1. Abstract syntax

approximation of f in α when $f^\# : \alpha(C) \rightarrow \alpha(C)$ and $\alpha \circ f \circ \alpha \leq f^\#$. $f^\alpha \stackrel{\text{def}}{=} \alpha \circ f \circ \alpha$ is the *best correct approximation* of f in α [11]. The abstraction α is *complete* when $\alpha \circ f = f^\alpha$ [10, 16]. The point-wise ordering on $uco(C)$ corresponds to the standard ordering used to compare abstract domains with regard to their precision: Let $\rho_i \in uco(C)$ and $A_i = \rho_i(C)$, then A_1 is more precise than A_2 (i.e., A_2 is an abstraction of A_1) iff $A_1 \sqsubseteq A_2$ in $uco(C)$. Let $\{A_i\}_{i \in I} \subseteq uco(C)$: $\sqcup_{i \in I} A_i$ is the most concrete among the domains in $uco(C)$ which is abstraction of all the A_i 's, i.e., $\sqcup_{i \in I} A_i$ is the *least* (w.r.t \sqsubseteq) *common abstraction* of all the A_i 's, and $\prod_{i \in I} A_i$ is the well-known *reduced product* of all the A_i 's, namely the most abstract among the domains in $uco(C)$ which is more concrete than every A_i . *Complementation* corresponds to the inverse of reduced product [7], namely an operator that, given two domains $C \sqsubseteq D$, gives as result the most abstract domain $C \ominus D$, whose reduces product with D is exactly C (i.e., $(C \ominus D) \sqcap D = C$). Therefore we have that $C \ominus D \stackrel{\text{def}}{=} \sqcup \{E \in uco(C) \mid D \sqcap E = C\}$. It has been proved in [15] that $\mathcal{M}(Mirr(C) \setminus Mirr(D)) = C \ominus D$.

Semantics. Given a transition system $\langle \Sigma, \curvearrowright \rangle$, where Σ is a nonempty set of states and $\curvearrowright \subseteq \Sigma \times \Sigma$ is the transition relation over states, we denote by Σ^+ and $\Sigma^\omega \stackrel{\text{def}}{=} \mathbb{N} \rightarrow \Sigma$ respectively the finite nonempty and the infinite sequences of symbols in Σ . Let $\sigma \in \Sigma^\infty \stackrel{\text{def}}{=} \Sigma^+ \cup \Sigma^\omega$ be a generic (finite or infinite) sequence of states then $|\sigma| \in \mathbb{N} \cup \omega$ is its length, σ_i is its i -th element, and σ_f is the final state if $\sigma \in \Sigma^+$. A finite (infinite) sequence of states σ is a program *trace* when for all $i < |\sigma|$: $\langle \sigma_i, \sigma_{i+1} \rangle \in \curvearrowright$, denoted $\sigma = \sigma_1 \widehat{\curvearrowright} \sigma_2 \widehat{\curvearrowright} \dots \widehat{\curvearrowright} \sigma_i \widehat{\curvearrowright} \dots$. In the following $\sigma_i \widehat{\curvearrowright} \sigma$ denotes the concatenation of a state σ_i with a trace σ . In this context μ is a *subtrace* of σ if there exists $i, j \in [0, |\sigma|)$, where $i < j$, and $\mu = \sigma_i \widehat{\curvearrowright} \dots \widehat{\curvearrowright} \sigma_j$. The *maximal trace semantics* of a transition system is $\tau^\infty \stackrel{\text{def}}{=} \tau^+ \cup \tau^\omega$, where τ^+ is the set of finite traces and τ^ω the set of infinite traces [9] of τ . From now on the trace semantics τ^∞ of the program P is considered as the concrete semantics $S[[P]] \subseteq \Sigma^\infty$. In [9] Cousot defines a hierarchy of semantics, where the big-step, termination and nontermination, Plotkin's natural, Smyth's demonic, Hoare's angelic relational and corresponding denotational, Dijkstra's predicate transformer weakest-precondition and weakest-liberal precondition and Hoare's partial and total axiomatic semantics, have all been derived by successive abstractions from the maximal trace semantics of a transition system τ^∞ . In this setting $uco(\wp(\Sigma^\infty))$ is the lattice of abstract semantics, namely each closure in $uco(\wp(\Sigma^\infty))$ represents an abstraction of trace semantics. For example the (*natural*) *denotational semantics* $\mathcal{D} \in uco(\wp(\Sigma^\infty))$ can be formalized as an ab-

Arithmetic expressions	$\mathbf{A}[n]\rho \stackrel{\text{def}}{=} n, \mathbf{A}[X]\rho \stackrel{\text{def}}{=} \rho(X)$
$\mathbf{A} : \mathcal{E}[\mathcal{X}] \rightarrow D_{\mathcal{T}} \text{ and } \text{var}[\mathbf{E}] \subseteq \mathcal{X}$	$\mathbf{A}[E_1 - E_2]\rho \stackrel{\text{def}}{=} \mathbf{A}[E_1]\rho - \mathbf{A}[E_2]\rho$
Boolean expressions	$\mathbf{B}[E_1 < E_2]\rho \stackrel{\text{def}}{=} \mathbf{A}[E_1]\rho < \mathbf{A}[E_2]\rho$
$\mathbf{B} : \mathcal{E}[\mathcal{X}] \rightarrow \{\text{true}, \text{false}, A\} \text{ and } \text{var}[\mathbf{B}] \subseteq \mathcal{X}$	$\mathbf{B}[B_1 \vee B_2]\rho \stackrel{\text{def}}{=} \mathbf{B}[B_1]\rho \vee \mathbf{B}[B_2]\rho$
	$\mathbf{B}[\neg B]\rho = \neg \mathbf{B}[B], \mathbf{B}[\text{true}/\text{false}]\rho \stackrel{\text{def}}{=} \text{true}/\text{false}$
Program actions	$\mathbf{S}[B]\rho \stackrel{\text{def}}{=} \{ \rho' \mid \mathbf{B}[B]\rho' = \text{true} \wedge \rho' = \rho \}$
$\mathbf{S} : \mathbf{A} \rightarrow (\mathcal{E}[\mathcal{X}] \rightarrow \wp(\mathcal{E}[\mathcal{X}])) \text{ and } \text{var}[\mathbf{A}] \subseteq \text{dom}(\rho)$	$\mathbf{S}[X := ?]\rho \stackrel{\text{def}}{=} \{ \rho' \mid \exists z \in \mathbb{Z} : \rho' = \rho[X := z] \}$
	$\mathbf{S}[X := E]\rho \stackrel{\text{def}}{=} \{ \rho[X := \mathbf{A}[E]\rho] \}$
	$\mathbf{S}[\text{true}]\rho \stackrel{\text{def}}{=} \mathbf{S}[\text{skip}]\rho \stackrel{\text{def}}{=} \{ \rho \}$

Table 2. Semantics

stract interpretation of τ^∞ : $\mathcal{D}(X) = \{ \sigma \in \Sigma^+ \mid \exists \delta \in X^+. \sigma_0 = \delta_0 \wedge \sigma_f = \delta_f \} \cup \{ \sigma \in \Sigma^\omega \mid \exists \delta \in X^\omega. \sigma_0 = \delta_0 \}$, where $X^+ \stackrel{\text{def}}{=} X \cap \Sigma^+$ and $X^\omega \stackrel{\text{def}}{=} X \cap \Sigma^\omega$. \mathcal{D} abstracts away the history of the computation by observing the input/output relation of finite traces and the input of diverging computations only. In the following we consider the simple imperative language defined in [12] with abstract syntax in Table 1, together with the following basic functions: $\text{lab}[L_1 : A \rightarrow L_2] \stackrel{\text{def}}{=} L_1$, $\text{lab}[P] \stackrel{\text{def}}{=} \{ \text{lab}[C] \mid C \in P \}$, $\text{var}[L_1 : A \rightarrow L_2] \stackrel{\text{def}}{=} \text{var}[A]$, $\text{var}[P] \stackrel{\text{def}}{=} \cup_{C \in P} \text{var}[C]$, $\text{succ}[L_1 : A \rightarrow L_2] \stackrel{\text{def}}{=} L_2$ and $\text{act}[L_1 : A \rightarrow L_2] \stackrel{\text{def}}{=} A$. Each variable $X \in \text{var}[P]$ has values in the semantic domain D , where the undefined value is denoted by \mathcal{T} and $D_{\mathcal{T}} \stackrel{\text{def}}{=} D \cup \{ \mathcal{T} \}$. We define an *environment* $\rho \in \mathcal{E}$ as a map from variables $X \in \text{dom}(\rho)$ to values $\rho(X) \in D_{\mathcal{T}}$. Let χ be a subset of variables, then $\rho|_\chi$ is the restriction of environment ρ to the domain $\text{dom}(\rho) \cap \chi$. The semantics is in Table 2. A *state* $\sigma_i \in \Sigma \stackrel{\text{def}}{=} \mathcal{E} \times \mathbb{C}$ is a pair $\langle \rho, C \rangle$, where C is the next command to be executed in the environment ρ . The *transition relation* between states specifies the set of states that can be reached from a given state: $R_P(\langle \rho, C \rangle) \stackrel{\text{def}}{=} \{ (\rho', C') \mid \rho' \in \mathbf{S}[\text{act}(C)]\rho, \text{succ}[C] = \text{lab}[C'], \rho, \rho' \in \mathcal{E}[P], C' \in P \}$. The *finite traces* of a program $P \in \mathbb{P}$ are obtained by the computation of the $\text{lf}p_{\subseteq} F[P]$, while the *infinite traces* by the computation of the $\text{gfp}_{\subseteq} F[P]$, where $F[P](X) \stackrel{\text{def}}{=} X \cup \{ \sigma_i \hat{\sim} \sigma'_i \hat{\sim} \sigma \mid \sigma'_i \in R_P(\sigma_i), \sigma'_i \hat{\sim} \sigma \in X \}$. We denote with $S[P]$ the set of finite and infinite traces of P .

Program Transformation. In [12] the authors define a language-independent methodology for systematically derive syntactic program transformations from semantic ones by mean of abstract interpretation (see Fig. 1). Given a program $P \in \mathbb{P}$ and a *syntactic transformation* $\tau : \mathbb{P} \rightarrow \mathbb{P}$ which returns the transformed program $\tau[P] \in \mathbb{P}$, we have that the corresponding *semantic transformation* t takes the semantics $S[P] \in \wp(\Sigma^\infty)$ of P , and returns the semantics $S[\tau[P]]$ of the transformed program. We consider programs as an abstraction of their semantics, as formalized by the Galois connection $(\langle \wp(\Sigma^\infty); \sqsubseteq \rangle, p, \langle \mathbb{P}; \leq \rangle, S)$, where $p[S]$ is the simplest program whose semantics upper approximates $S \in \wp(\Sigma^\infty)$ [12]. Therefore the *semantic transformation* $t : \wp(\Sigma^\infty) \rightarrow \wp(\Sigma^\infty)$ induced by the *syntactic transformation* τ is $t(S[P]) = S[\tau[p(S[P])]]$, while the *syntactic*

transformation τ induced by the semantic transformation t is $\tau[P] = p(t(S[P]))$. A program transformation is *correct* if at some level of abstraction it is meaning preserving, namely a syntactic transformation τ is correct w.r.t. an observational abstraction $\alpha_{\mathcal{O}}$ if $\alpha_{\mathcal{O}}(S[P]) = \alpha_{\mathcal{O}}(S[\tau[P]])$ [12].

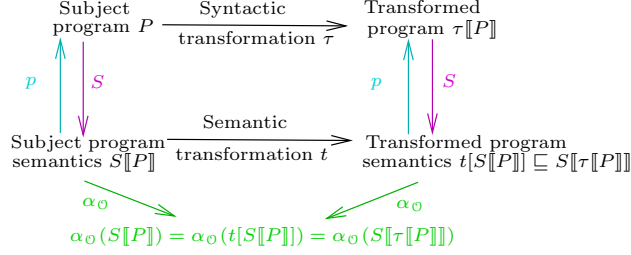


Fig. 1. Syntactic-Semantic Program Transformations

3 Semantic-based program obfuscators

Traditionally a program transformation $\tau : \mathbb{P} \rightarrow \mathbb{P}$ is an obfuscation if: 1) it is a potent transformation and 2) P and $\tau[P]$ have the same observational behavior, *i.e.*, if P fails to terminate or it terminates with an error condition then $\tau[P]$ may or may not terminate; otherwise $\tau[P]$ must terminate and produce the same output as P [4–6]. This definition requires that given an initial state σ_0 , if $\sigma_0 \hat{\sigma} \sigma \in S[P]^\omega$ or $\sigma_0 \hat{\sigma} \sigma \in S[P]^+$, where $\sigma_e \in Err$ with Err being the set of error states, then $\sigma_0 \hat{\sigma} \sigma' \in S[\tau[P]]^\infty$; otherwise if $\sigma \in S[P]^+$ then $\sigma \in \mathcal{D}(S[\tau[P]]^+)$. We denote by \mathcal{C} the family of these code transformations.

3.1 Abstract interpretation-based code obfuscation

In this section we specify what is hidden by a syntactic transformation, by introducing a novel definition of code obfuscation based on semantics.

Definition 1. Let $\tau : \mathbb{P} \rightarrow \mathbb{P}$ be a program transformation. τ is potent if there is a property $\alpha \in uco(\wp(\Sigma^\infty))$ such that: $\alpha(S[P]) \neq \alpha(S[\tau[P]])$.

Therefore a transformation τ is potent when there exists a property α and a program P such that the approximation of the concrete trace semantics of P is different from the same approximation for the concrete trace semantics of $\tau[P]$.

Example 1. Let us consider a program P and its transformed version P' :

<p>P:</p> <p>L₁: X:= ? → L₂</p> <p>L₂: Y:= ? → L₃</p> <p>L₃: out:= (X - Y)² → J</p>	<p>P':</p> <p>L₁: X:= ? → L₂</p> <p>L₂: Y:= ? → L₃</p> <p>L₃: out:= X² → L₄</p> <p>L₄: out:= out - 2XY → L₅</p> <p>L₅: out:= out - Y² → J</p>
---	--

Let us define $\alpha \in uco(\wp(\Sigma^\infty))$ as $\alpha(X) \stackrel{\text{def}}{=} \{ \sigma \in \Sigma^+ \mid \exists \eta \in X : |\sigma| = |\eta| \}$. The property α observes the length of program traces. In this case we have $\alpha(S[[P]]) = \{ \sigma \in \Sigma^+ \mid |\sigma| = 3 \}$, while $\alpha(S[[\tau[[P]]]]) = \{ \sigma \in \Sigma^+ \mid |\sigma| = 5 \}$. It is clear that $\alpha(S[[P]]) \neq \alpha(S[[\tau[[P]]]])$, *i.e.*, the transformation obfuscates α .

In order to factor the observational semantics into preserved and masked properties, we define the most concrete property preserved by a transformation $\tau : \mathbb{P} \rightarrow \mathbb{P}$, as follows $\delta_\tau \stackrel{\text{def}}{=} \sqcap \{ \varphi \in uco(\wp(\Sigma^\infty)) \mid \varphi(S[[P]]) = \varphi(S[[\tau[[P]]]]) \}$.

Lemma 1. *Given a transformation $\tau : \mathbb{P} \rightarrow \mathbb{P}$, $\delta_\tau(S[[P]]) = \delta_\tau(S[[\tau[[P]]]])$.*

Given a program transformation τ we want to characterize the set of properties that are not preserved, *i.e.*, obfuscated, by τ . By considering the transformation τ and the property $\alpha \in uco(\wp(\Sigma^\infty))$ that the attacker wants to observe, we note that $\alpha \ominus (\delta_\tau \sqcup \alpha)$ is precisely what the transformation τ hides of the property α . In fact when $\alpha \ominus (\delta_\tau \sqcup \alpha) \neq \top$ some parts of the property α has been lost in the transformation. In this case we say that the property α is *obfuscated* by the transformation τ , because that property cannot be observed on the semantics of the transformed program.

Definition 2. $O_{\delta_\tau} = \{ \alpha \in uco(\wp(\Sigma^\infty)) \mid \alpha \ominus (\delta_\tau \sqcup \alpha) \neq \top \}$ is the set of properties obfuscated by $\tau : \mathbb{P} \rightarrow \mathbb{P}$.

Given $\delta \in uco(\wp(\Sigma^\infty))$, we define a δ -obfuscator as any potent program transformation $\tau : \mathbb{P} \rightarrow \mathbb{P}$, such that every program is equivalent to its obfuscated version w.r.t. the particular observational semantics δ .

Definition 3. $\tau : \mathbb{P} \rightarrow \mathbb{P}$ is a δ -obfuscator if $\delta = \delta_\tau$ and $O_\delta \neq \emptyset$

Lemma 2. *Given a δ -obfuscator $\tau : \mathbb{P} \rightarrow \mathbb{P}$, then for each $\alpha \in O_\delta$ there exists a program P such that $\alpha(S[[P]]) \neq \alpha(S[[\tau[[P]]]])$.*

The above definition of obfuscator is a generalization of the classical one by Collberg *et al.*, introduced at the beginning of this Section.

Theorem 1. *If τ is a \mathcal{D} -obfuscator then $\tau \in \mathcal{C}$.*

In particular we can define a *partial order* between obfuscating transformations, by considering the set of properties that the transformations hide. Given two program transformations τ and τ' , then τ' is more potent than τ , denoted $\tau \leq_P \tau'$, if $O_{\delta_\tau} \subseteq O_{\delta_{\tau'}}$. We can also compare the potency of τ and τ' w.r.t. a particular property α , namely by measuring the approximation in the knowledge of that property that can be obtained by observing semantics. Let $\alpha \in O_{\delta_\tau} \cap O_{\delta_{\tau'}}$, then τ' is more potent than τ w.r.t. α , denoted $\tau \leq_\alpha \tau'$, if $\alpha \ominus (\delta_{\tau'} \sqcup \alpha) \sqsubseteq \alpha \ominus (\delta_\tau \sqcup \alpha)$. In this case τ' obfuscates the property α more than what τ does. From the structure of the lattice of abstract interpretations $uco(\wp(\Sigma^\infty))$ we can derive the some basic properties of O_δ and therefore of the potency of code obfuscations.

Proposition 1. *Let $\delta, \mu \in uco(\wp(\Sigma^\infty))$ and τ, τ' be program transformations:*
1) $O_\delta = \{ \alpha \in uco(\wp(\Sigma^\infty)) \mid \alpha \notin \uparrow \delta \}$; 2) *If $\mu \sqsubseteq \delta$ then $O_\mu \subset O_\delta$, *i.e.*, $\tau_\mu \leq_P \tau_\delta$;*
3) $O_{\delta \sqcup \mu} = O_\delta \cup O_\mu$; 4) *If $\tau \leq_\delta \tau'$, then for each property $\mu \sqsubseteq \delta$ we have $\tau \leq_\mu \tau'$;*
5) *If $\tau \leq_{\delta \sqcap \mu} \tau'$, then $\tau \leq_\delta \tau'$ and $\tau \leq_\mu \tau'$.*

3.2 Constructive characterization of potency

In this section we define a method for deriving δ_τ for a given syntactic transformation $\tau : \mathbb{P} \rightarrow \mathbb{P}$. By Definition 2, this provides a characterization of the potency of a code transformation. Let $t : \wp(\Sigma^\infty) \rightarrow \wp(\Sigma^\infty)$ be a semantic transformation and $K_{P,t} : uco(\wp(\Sigma^\infty)) \rightarrow uco(\wp(\Sigma^\infty))$ be a domain transformer that, given a property $\mu \in uco(\wp(\Sigma^\infty))$, returns the closest abstraction preserved by t on $P \in \mathbb{P}$: $K_{P,t} \stackrel{\text{def}}{=} \lambda\mu. \sqcap \{ \varphi \in uco(\wp(\Sigma^\infty)) \mid \mu \sqsubseteq \varphi \wedge \varphi(S[[P]]) = \varphi(t(S[[P]])) \}$. For a given $\mu \in uco(\wp(\Sigma^\infty))$, $K_{P,t}(\mu)$ is a closure operator on sets of traces. In order to characterize the set of its fix-points, we have to specify the set of traces $X \subseteq \Sigma^\infty$ preserved by the transformation t on a particular program P . The predicate $Pres_{P,t}(X)$, where $X \in \wp(\Sigma^\infty)$, precisely captures this notion, in fact $Pres_{P,t}(X)$ evaluates to *true* if and only if $\forall Y \subseteq S[[P]] : Y \subseteq X \Rightarrow t(Y) \subseteq X$.

Lemma 3. $\{ X \in \wp(\Sigma^\infty) \mid Pres_{P,t}(X) \}$ is preserved by t on program P .

Theorem 2. $K_{P,t}(id) = \{ X \in \wp(\Sigma^\infty) \mid Pres_{P,t}(X) \}$

$K_{P,t}(id)$ models the most concrete property preserved by the transformation t for the program P . The generalization of this notion for all programs follows straightforwardly.

Corollary 1. Let $\tau : \mathbb{P} \rightarrow \mathbb{P}$, then $\delta_\tau = \bigsqcup_{P \in \mathbb{P}} K_{P, \rho \circ \tau \circ S}(id)$.

4 An example: Obfuscation by program specialization

In this section we consider constant propagation as code obfuscation. This proves that our semantic-based approach to code obfuscation is adequate both to include a wide range of program transformation techniques and to compare them by extracting what is actually masked by the transformation. We follow Cousot & Cousot [12] in the definition of an algorithm for constant propagation.

4.1 Cousot's constant propagation

The *residual* $R[[E]]\rho$ of an arithmetic or boolean expression E in an environment ρ is the expression resulting by the specialization of E in such environment (see Table 3). An expression $E \in \mathbb{E} \cup \mathbb{B}$ is *static* in the environment ρ , denoted $\mathbf{static}[[E]]\rho$, if it can be fully evaluated in ρ , that is $\mathit{var}[[E]] \subseteq \mathit{dom}(\rho)$; otherwise E is *dynamic*. With $R[[A]]\rho$ we denote the specialization of action A in the environment ρ (see Table 3). We consider syntactic and semantic program transformations relative to the constant propagation problem as formalized in [12], where the arguments of the program transformation are the semantics $S[[P]]$ and the result of a constant detection static analysis $S^C[[P]]$.

Abstract semantics: It is defined as $S^C = \alpha^C \circ S$, where α^C is given by:

$$\alpha^C(S[[P]]) = \lambda L. \lambda X. \bigsqcup \{ \rho(X) \mid \sigma \in S[[P]], C \in \mathbb{C}, \sigma_i = \langle \rho, C \rangle, \mathit{lab}[[C]] = L \}$$

<u>Expressions</u>	$R \in \mathbb{E} \times \mathcal{E} \rightarrow \mathbb{E}$
	$R[\rho] = n$
	$R[X] = \text{if } X \in \text{dom}(\rho) \text{ then } \rho(X) \text{ else } X$
	$R[E_1 - E_2] = \text{let } E_1^r = R[E_1]\rho \text{ and } E_2^r = R[E_2]\rho$
	$\text{if } E_1^r = \Lambda \text{ or } E_2^r = \Lambda \text{ then } \Lambda$
	$\text{else if } E_1^r = n_1 \text{ and } E_2^r = n_2 \text{ then } n = n_1 - n_2 \text{ else } E_1^r - E_2^r$
	$R \in \mathbb{B} \times \mathcal{E} \rightarrow \mathbb{B}$
	$R[E_1 < E_2] = \text{let } E_1^r = R[E_1]\rho \text{ and } E_2^r = R[E_2]\rho$
	$\text{if } E_1^r = \Lambda \text{ or } E_2^r = \Lambda \text{ then } \Lambda$
	$\text{else if } E_1^r = n_1 \text{ and } E_2^r = n_2 \text{ and } b = n_1 < n_2$
	$\text{then } b \text{ else } E_1^r < E_2^r$
	$R[B_1 \vee B_2] = \text{let } B_1^r = R[B_1]\rho \text{ and } B_2^r = R[B_2]\rho$
	$\text{if } B_1^r = \Lambda \text{ or } B_2^r = \Lambda \text{ then } \Lambda$
	$\text{else if } B_1^r = \text{true} \text{ or } B_2^r = \text{true} \text{ then } \text{true}$
	$\text{else if } B_1^r = \text{false} \text{ then } B_2^r$
	$\text{else if } B_2^r = \text{false} \text{ then } B_1^r$
	$\text{else } B_1^r \vee B_2^r$
	$R[\neg B] = \text{let } B^r = R[B]\rho$
	$\text{if } B^r = \Lambda \text{ then } \Lambda$
	$\text{else if } B^r = \text{true} \text{ then } \text{false}$
	$\text{else if } B^r = \text{false} \text{ then } \text{true}$
	$\text{else } \neg B^r$
	$R[\text{true}] = \text{true}$
	$R[\text{false}] = \text{false}$
<u>Actions</u>	$R \in \mathbb{A} \times \mathcal{E} \rightarrow \mathbb{A}$
	$R[B] = \langle \rho, R[B]\rho \rangle$
	$R[X := ?] = \langle \rho \setminus X, X := ? \rangle$
	$R[X := E] = \text{if } \mathbf{static}[E]\rho \text{ then } \langle \rho[X := R[E]\rho], \text{skip} \rangle$
	$\text{else } \langle \rho \setminus X, X := R[E]\rho \rangle$

Table 3. Expression and Action specialization

This function, given a label L and a variable X , returns the least upper bound \sqcup on the flat lattice of program values for X at the program point L .

Semantic transformation: The semantic transformation t^C is defined in [12] where: $t^C[S[P], S^C[P]] \stackrel{\text{def}}{=} \{t^C[\sigma, S^C[P]] \mid \sigma \in S[P]\}$ is the transformation of semantics, $t^C[\sigma, S^C[P]] \stackrel{\text{def}}{=} \lambda i. t^C[\sigma_i, S^C[P]]$ is the transformation of traces, and $t^C[\langle \rho, C \rangle, S^C[P]] \stackrel{\text{def}}{=} \langle \rho, t^C[C, \tau^C(\text{lab}[C])] \rangle$ is the transformation of states, where $t^C[L_1 : A \rightarrow L_2, \rho^C] \stackrel{\text{def}}{=} L_1 : t^C[A, \rho^C] \rightarrow L_2$ is command specialization and $t^C[A, \rho^C] = \text{let } \langle \rho_r, A_r \rangle \stackrel{\text{def}}{=} R[A]\rho \mid \{X \in \mathbb{X} \mid \rho^C(X) \in D_r\} \in A_r$ is action specialization. The syntactic transformation τ^C can be systematically derived as shown in [12].

Observational semantics: Given a partial trace σ , the observational semantics α_0^C returns the sequence of its environments: $\alpha_0^C(S[P]) \stackrel{\text{def}}{=} \{\alpha_0^C(\sigma) \mid \sigma \in S[P]\}$, $\alpha_0^C(\sigma) \stackrel{\text{def}}{=} \lambda i. \alpha_0^C(\sigma_i)$, and $\alpha_0^C(\langle \rho, C \rangle) \stackrel{\text{def}}{=} \rho$.

a:= 1; b:=2; c:=3; d:=3; e:=0;	$L_1 : a:= 1; b:=2; c:=3; d:=3; e:=0; \rightarrow L_2$
while B do	$L_2 : B \rightarrow L_3$
	$L_2 : \neg B \rightarrow L_5$
b:=2*a; d:=d+1; e:=e-a;	$L_3 : b:=2*a; d:=d+1; e:=e-a; \rightarrow L_4$
a:=b-a; c:=e+d;	$L_4 : a:=b-a; c:=e+d; \rightarrow L_2$
endw	$L_5 : \text{stop} \rightarrow /$

Table 4. A simple program from [8]

4.2 Code obfuscation by constant propagation

In order to specify the properties obfuscated by constant propagation τ^C , we derive δ_{τ^C} . Let us define the property θ as follows: $\theta(S[P]) \stackrel{\text{def}}{=} \{\theta(\sigma) | \sigma \in S[P]\}$, $\theta(\sigma) \stackrel{\text{def}}{=} \lambda i. \theta(\sigma_i)$, and $\theta(\langle \rho, C \rangle) \stackrel{\text{def}}{=} \langle \rho, \text{lab}[C], \text{succ}[C] \rangle$.

Theorem 3. $\theta = \delta_{\tau^C}$ and $O_\theta \neq \emptyset$.

It is worth noting that θ is the most concrete property preserved by constant propagation τ^C . Let us consider the property $\vartheta \in \text{uco}(\wp(\Sigma^\infty))$, observing the environment, the labels, and the type of actions: $\vartheta(\tau^\infty) \stackrel{\text{def}}{=} \{\vartheta(\sigma) | \sigma \in \tau^\infty\}$, $\vartheta(\sigma) \stackrel{\text{def}}{=} \lambda i. \vartheta(\sigma_i)$, and $\vartheta(\langle \rho, C \rangle) \stackrel{\text{def}}{=} \langle \rho, \text{lab}[C], \text{succ}[C], \text{type}(\text{act}[C]) \rangle$, where type maps actions into the following set of action types $\{\text{assign}, \text{skip}, \text{test}\}$. This property belongs to O_θ , meaning that $O_\theta \neq \emptyset$, as requested by Definition 3 and shown in Example 2 below. This proves that τ^C is actually a program obfuscator, hiding the type of the command actions. This is a consequence of the structure of the masked closure $\vartheta \ominus (\vartheta \sqcup \theta)$, which characterizes what is obfuscated from ϑ by τ^C . Consider the closure η which observes the type of actions defined as follows:

$$\eta = \lambda X. \left\{ \sigma \mid \begin{array}{l} \sigma' \in X \text{ and } \forall i. \sigma_i = \langle \rho_i, C_i \rangle, \sigma'_i = \langle \rho'_i, C'_i \rangle \\ \text{type}(C_i) = \text{type}(C'_i) \end{array} \right\}$$

The following theorem specifies that η is masked by the constant propagation, *i.e.*, τ^C obfuscates η .

Theorem 4. $\vartheta \ominus (\vartheta \sqcup \theta) \sqsubseteq \eta$.

Example 2. As observed above, ϑ is not preserved by t^C , namely it could happen that: $\vartheta(S[P]) \neq \vartheta(t^C[S[P], S^C[P]])$. In the following we represent the environment as a tuple $(v_a, v_b, v_c, v_d, v_e)$ of values corresponding to the variables a, b, c, d, e in a certain execution point. Let us run the program in Table 4, and consider the states $\sigma_2 = \langle (1, 2, 3, 3, 0), L_3 : b := 2*a; d := d+1; e := e-a; \rightarrow L_4 \rangle$ and $\sigma_3 = \langle (1, 2, 3, 4, -1), L_4 : a := b-a; c := e+d \rightarrow L_2 \rangle$. Their transformed versions are: $t^C(\sigma_2) = \langle (1, 2, 3, 3, 0), L_3 : d := d+1; e := e-a; \rightarrow L_4 \rangle$ and $t^C(\sigma_3) = \langle (1, 2, 3, 4, -1), L_4 : \text{skip} \rightarrow L_2 \rangle$. In this case $\vartheta(\sigma_2) = \langle (1, 2, 3, 3, 0), L_3, L_4, \text{assign} \rangle$ and $\vartheta(\sigma_3) = \langle (1, 2, 3, 4, -1), L_4, L_2, \text{assign} \rangle$; while $\vartheta(t^C(\sigma_2)) = \langle (1, 2, 3, 3, 0), L_3, L_4, \text{assign} \rangle$ and $\vartheta(t^C(\sigma_3)) = \langle (1, 2, 3, 4, -1), L_4, L_2, \text{skip} \rangle$, showing that the property η is not preserved.

5 Discussion

In this paper we introduce a notion of code obfuscation providing a general enough definition including both most program transformation techniques as obfuscators and the standard definition of Collberg *et al.* [5] as special cases. Moreover, it provides advanced techniques for comparing obfuscating algorithms relatively to their potency in the lattice of abstract interpretations. This definition can be considered as the first step towards a semantic-based theory for code obfuscation. Note however that in Definition 1 we consider abstractions of concrete semantics and not arbitrary sound, possibly incomplete, abstract interpretations. This makes Definition 1 too strong for modeling attackers which can be any decidable sound approximation of semantics, *e.g.*, arbitrary static program analyzers. Consider the following example of program transformation:

$$\begin{array}{ll}
 P: & P': \\
 L_1: X := ? \rightarrow L_2 & L_1: X := ? \rightarrow L_2 \\
 L_2: Y := ? \rightarrow L_3 & L_2: Y := ? \rightarrow L_3 \\
 L_3: \text{out} := (X - Y)^2 \rightarrow I & L_3: \text{out} := X^2 - 2XY + Y^2 \rightarrow I
 \end{array}$$

Let's consider the property of the sign of a variable, *i.e.*, the abstract domain $Sign \stackrel{\text{def}}{=} \{\mathbb{Z}, \mathbb{Z}^+, \mathbb{Z}^-, \emptyset\}$. By considering the sign of the variable `out`, when `X` is positive and `Y` is negative, we have that: $Sign(S[P]) = Sign(S[\tau[P]]) = \mathbb{Z}^+$, while $S^{Sign}[P] = \mathbb{Z}^+$ and $S^{Sign}[\tau[P]] = \mathbb{Z}$. Therefore the previous transformation obfuscates program P for the sign analysis, because the static sign analysis of the transformed program is unable to get the sign of `out`. This is not captured by Definition 1, because $Sign$ is incomplete for integer addition [16]. Indeed, in Definition 1, the abstraction is applied to the concrete semantics, returning \mathbb{Z}^+ in both cases. It would be important to weaken our notion of potency by considering abstract semantics derived by sound (possibly incomplete) approximations of the concrete semantics. This is crucial in order to include in our model of attackers arbitrary sound program analyzers. Moreover the systematic design of program transformation by abstract interpretation [12] can be applied for the systematic design of code obfuscation algorithms, driven by the abstraction (semantic property) to be masked. We can observe that other program transformations, such as *abstract watermarking* [13], can be seen as a particular code obfuscation. The relation between our notion of code obfuscation and abstract watermarking deserves further investigation.

References

1. B. Barak, O. Goldreich, R. Impagliazzo, and S. Rudich. On the (Im)possibility of Obfuscating Programs. In *Advances in Cryptology, Proc. of Crypto'01*, 2001, volume 2139 of LNCS, pages 1-18. Springer-Verlag.
2. C. Collberg and C. Thomborson. Watermarking, Tamper-Proofing, and Obfuscation-Tools for Software Protection. In *IEEE Trans. Software Eng.*, pages 735-746, 2002.
3. S. Chandrasekharan and S. Debray. Deobfuscation: Improving Reverse Engineering of Obfuscated Code. Draft, 2005.

4. C. Collberg and C. Thomborson. Breaking Abstractions and Unstructural Data Structures. In *Proc. of the 1994 IEEE Internat. Conf. on Computer Languages (ICCL '98)*, pages 28-37, 1998.
5. C. Collberg, C. Thomborson, and D. Low. A Taxonomy of Obfuscating Transformations. Technical Report 148, Dept. of Computer Science, The Univ. of Auckland, 1997.
6. C. Collberg, C. Thomborson, and D. Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages (POPL '98)*, pages 184-196. ACM Press, 1998.
7. A. Cortesi, G. Filé, R. Giacobazzi, C. Palamidessi, and F. Ranzato. Complementation in abstract interpretation. In *ACM Trans. Program. Lang. Syst.*, 19(1):7-47, 1997.
8. P. Cousot. Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes. PhD Thesis, Université Scientifique et Médicale de Grenoble, Grenoble, France, 1978.
9. P. Cousot. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *Theoretical Computer Science*, 277(1-2):47-103, 2002.
10. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symp. on Principles of Programming Languages (POPL '77)*, pages 238-252. ACM Press, New York, 1977.
11. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the 6th ACM Symp. on Principles of Programming Languages (POPL '79)*, pages 269-282. ACM Press, New York, 1979.
12. P. Cousot and R. Cousot. Systematic Design of Program Transformation Frameworks by Abstract Interpretation. In *Conference Record of the Twentyninth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 178-190, New York, NY, 2002. ACM Press.
13. P. Cousot and R. Cousot. An Abstract Interpretation-Based Framework for Software Watermarking. *Conference Record of the Thirtyfirst Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 173-185. ACM Press, New York, NY, 2004.
14. L. D'Anna, B. Matt, A. Reisse, T. Van Vleck, S. Schwab, and P. LeBlanc. Self-Protecting Mobile Agents Obfuscation Report. Technical report, Network Associates Laboratory, 2003.
15. G. Filé and F. Ranzato. Complementation of abstract domains made easy. In *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming (JICSLP '96)*, pages 348-362. The MIT Press, Cambridge, Mass., 1996.
16. R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *J. of the ACM.*, 47(2):361-416, 2000.
17. J. Morgado. Some results on the closure operators of partially ordered sets. *Portug. Math.*, 19(2):101-139, 1960.
18. R. Paige. Future directions in program transformations. In *ACM SIGPLAN Not.*, volume 32, pages 94-97, 1997.